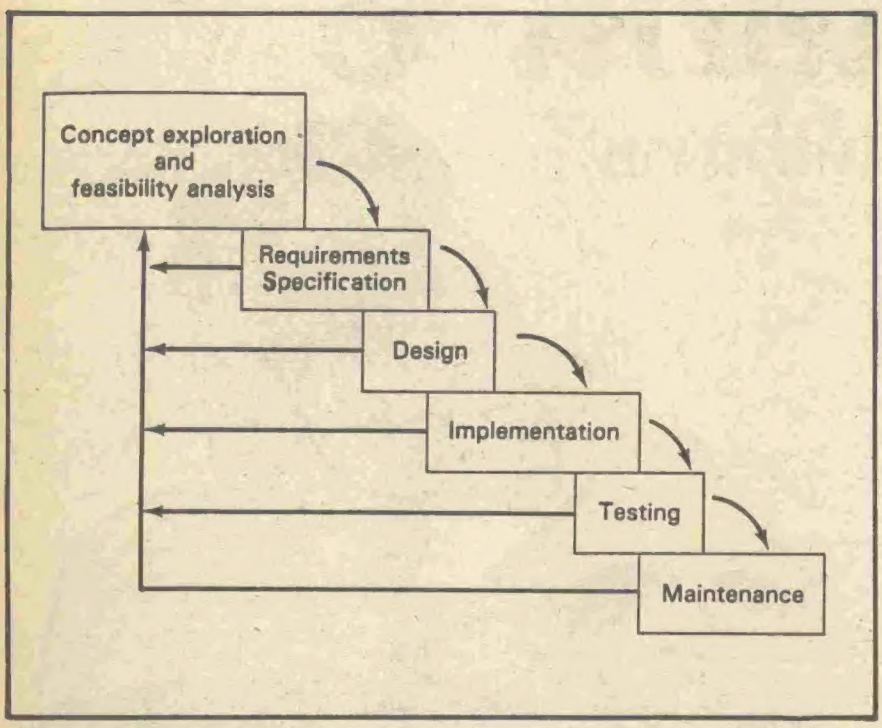
เขียน C ให้อ่านง่าย

กับการเขียนโปรแกรมที่คุณต้องการ ด้วยประสิทธิภาพอันเต็มเปี่ยม เตรียมพร้อมกับการพัฒนาระบบในอนาคต

หลายคนคงเคยเขียนโปรแกรมมาบ้างแล้ว ถ้าเขียนด้วยภาษาระดับสูงแล้วได้ออปเจกต์โค้ดขนาด 40-60 กิโลไบต์ ก็สามารถเขียนเสร็จได้ภายในวันหรือสองวัน และนำมาใช้กับงานตามที่ต้องการได้ หลังจากที่ใช้ได้แค่ปีเดียวแล้วต้องการที่จะพัฒนาต่อให้ดียิ่งขึ้น ก็รื้อโปรแกรมนั้นมาอ่านเพื่อจะได้ทำต่อ แต่ปรากฏว่าเราอ่านไม่รู้เรื่องเลย ตัวแปรก็มีแต่ a,b,c,i,m คอมเมนต์ (comment) ก็ไม่มี แต่ละฟังก์ชันก็ไม่รู้ว่าทำอะไร แถมยังมีการกระโดดไปมาด้วยคำสั่ง goto อีกด้วย เมื่อนั่งปวดหัวอยู่ได้สักพัก ก็นึกเสียดายว่าทำไมตอนนั้นถึงได้เขียนอะไรอย่างนั้น ว่าแล้วก็ลงมือเขียนใหม่หมดเลย คราวนี้เมื่อเขียนไปได้กว่า 100 กิโลไบต์ ถ้ายังเขียนด้วยนิสัยเก่า ๆ คุณจะรู้สึกว่างานมันไม่เดินแล้ว เพราะการเขียนที่ทำให้คนอ่านอ่านยาก และรวมทั้งคนเขียนเองด้วย ยิ่งเขียนก็ยิ่งรู้สึกว่าโครงสร้างของโปรแกรมรุ่นวายไปหมด บัก (bug) ก็มีเยอะมาก ในที่สุดก็เขียนไม่เสร็จตามที่ตั้งใจไว้

ความจริงถ้าเขียนโปรแกรมให้อ่านง่ายกว่านี้สักหน่อย มีคอมเมนต์ที่ดี แม้จะลำบากสักนิดในตอนเริ่มหัด แต่ก็จะเห็นคุณค่ามากขึ้นเมื่อโปรแกรมโตขึ้น ว่าช่วยให้เขียนง่ายขึ้น เร็วขึ้น เข้าใจง่าย แก้ไขเพิ่มเติมก็ง่ายบักก็น้อย

เมื่อคิดจะทำซอฟต์แวร์สักชิ้นหนึ่งจะประกอบด้วยขั้นตอนอะไรบ้าง หลายคนก็บอกว่า เริ่มต้นด้วยการคิดว่าจะทำอะไร ทำอย่างไร เมื่อคิดออกแล้วก็ลงมือเขียน เขียนไปแก้บักไป เสร็จแล้วก็เอาไปใช้งานได้ เป็นอันเรียบร้อย แต่หลักการทำซอฟต์แวร์ที่ดีแล้ว มีหลักวิชาการกล่าวถึงขั้นตอนสำคัญ ๆ ไว้ด้วยว่าประกอบด้วย 5 ขั้นดังในรูปที่ 1



รูปที่ 1 ผังแสดงช่วงอายุขัยของ software (Software Life Cycle)

1. วิเคราะห์หลักการและความเป็นไปได้ เช่น ทำแล้วจะมีประโยชน์หรือเปล่า ทำแล้วจะเสร็จหรือไม่ ถ้าเป็นซอฟต์แวร์เพื่อตั้งใจไว้ขายก็ต้องคิดด้วยว่าต้องใช้คนกี่คน ทำให้เสร็จภายในเวลาเท่าใด ทำแล้วขายออกหรือเปล่า กำไรหรือขาดทุนแค่ไหน เป็นต้น

2. กำหนดขอบเขตของงาน วิเคราะห์และกำหนดขอบเขตว่างานที่จะทำนั้น จะทำอะไรได้บ้าง

3. ออกแบบ เป็นการออกแบบทั้งในระดับกว้างหรือทั้งระบบ (System Design) และในระดับรายละเอียด (Detail Design)

4. เขียนโปรแกรม รวมทั้งแก้บักและทำเอกสารประกอบโปรแกรมด้วย

5. ทดสอบ ทดสอบดูว่าโปรแกรมที่ทำขึ้นมาตรงตามขอบเขตที่กำหนดไว้ในตอนแรกหรือไม่

6. ใช้และบำรุงรักษา คือนำออกใช้และปรับปรุงเพิ่มเติมตามความต้องการของผู้ใช้ที่อาจมีขึ้นในภายหลัง

ขั้นตอนต่าง ๆ เหล่านี้จะช่วยให้ได้โปรแกรมที่มีคุณภาพ, ถูกต้อง, มีประสิทธิภาพ, อ่านง่ายและนำกลับมาใช้งานใหม่ได้ง่ายด้วย ซึ่งเป็นขั้นตอนที่ใช้ได้ทั่วไปกับทุกภาษา ถ้าเป็นภาษา C ซึ่งเต็มไปด้วยสัญลักษณ์มากมายที่ทำให้เสี่ยงต่อการทำให้โปรแกรมทำงานผิดจากที่ต้องการได้ บทความนี้จะขอกล่าวถึงเฉพาะการเขียนภาษา C ให้อ่านง่าย คำว่าอ่านง่ายนี้ คือ ผู้อ่านจะต้องอ่านซอร์สโค้ดได้รวดเร็วและทำความเข้าใจได้ง่าย ว่าผู้เขียนเขียนแต่ละส่วนนั้นเพื่อทำอะไรทำอย่างไร ซึ่งมีความสำคัญมาก เพราะตลอดช่วงอายุขัยของซอฟต์แวร์ทั้ง 6 ช่วงที่กล่าวมาข้างต้นนั้น ซอร์สโค้ดจะต้องถูกอ่านซ้ำแล้วซ้ำอีกตั้งแต่เริ่มเขียน, ตรวจสอบ, แก้บัก, ทดสอบและบำรุงรักษา

ตั้งชื่อให้ดี

ส่วนนี้สำคัญมากตั้งแต่การตั้งชื่อตัวแปร, ฟังก์ชัน, ชนิดของข้อมูล, ตัวคงที่, มาโครและอื่น ๆ ที่ถูกต้องและดีถือเป็นหลักการขั้นต้นของการเขียนโปรแกรมให้อ่านง่าย ชื่อจะต้องสื่อให้เห็นการใช้งานและประโยชน์ของสิ่งของนั้น ๆ ให้ได้ เช่นสมมติจะเขียนฟังก์ชันที่ทำหน้าที่ค้นหาข้อมูลแบบ binary search ก็อย่าไปตั้งชื่อเป็น s( ) หรือ f( ) เลย

ลองดูตัวอย่างที่ยกมาในรูปที่ 2.1 จะเห็นว่ามีแต่ชื่อ a,b,c,... ซึ่งคนอ่านจะต้องเสียเวลามาไล่ตัวฟังก์ชันดูว่าฟังก์ชันนี้ทำอะไร แต่พอดูในรูป 2.2 แล้ว ก็เข้าใจทันทีว่าในรูป 2.2 นี้คือฟังก์ชันสำหรับการทำ binary search ที่จริงในรูป 2.1 ก็คือฟังก์ชันเดียวกับ binary search ในรูป 2.2 เพียงแต่ตั้งชื่อเปลี่ยนไปให้ดูยากขึ้น แค่นี้ก็จะเห็นได้ว่าการตั้งชื่อที่ไม่เหมาะ สามารถทำให้อัลกอริทึมง่าย ๆ กลายเป็นยากไปได้ ในตอนท้ายของรูป 2.1 จะพบ −1 คืออะไร ถ้าอ่านดูในรูป 2.2 จะเห็น NOT_FOUND ซึ่งถูกกำหนดไว้ก่อนแล้วว่าเป็น −1 จะทำให้คุณเข้าใจได้ทันทีว่าฟังก์ชันนี้จะรีเทอร์นค่าเป็น NOT_FOUND หรือ −1 ให้หากีย์ที่ให้มาหาไม่เจอในลิสต์

สำหรับการตั้งชื่อที่ดีนั้นหลายคนมีหลักการต่าง ๆ กันออกไป ซึ่งบางข้อก็คล้าย ๆ กัน บางข้อก็ต่างกัน บางข้อก็ถึงกับขัดกันเลยก็มี เช่นบางคนบอกว่าชื่อตัวแปรให้ตัวใหญ่บ้างเล็กบ้าง แต่บางคนบอกว่าให้ใช้ตัวเล็กหมด แต่ที่เป็นสากลนิยมหลายคนใช้จะเป็นอย่างนี้ (ในที่นี้จะใช้คำว่า 'ควร' หรือ 'อาจจะ' เท่านั้น ไม่ใช้คำว่า 'ต้อง')

```
int f (a, b, c, d)
int a, b, c, *d;
{
    int e;
    do
    {
        e = (b + c) / 2;
        if (a < d[e])
          c = e - 1;
        else
          b = e + 1;
    }
    while ( (a ! = d[e]) && (b < = c));
    return ( (a = = d[e]) ? e : -1);
}
```

รูปที่ 2.1 ตัวอย่างการเขียนฟังก์ชันที่ตั้งชื่อไม่สื่อความหมาย ทำให้เข้าใจยาก

```
int Binary_Search (key, lo, hi, list)
int key, lo, hi, *list;
{
    int mid;
    do
    {
        mid = (lo + hi) / 2;
        if (key < list[mid])
          hi = mid - 1;
        else
          lo = mid + 1;
    }
    while ( (key ! = list[mid]) && (lo < = hi));
    return ( (key = = list[mid]) ! mid : NOT_FOUND);
}
```

รูปที่ 2.2 ตัวอย่างการเขียนฟังก์ชันโดยตั้งชื่อให้สื่อความหมาย ซึ่งช่วยให้รู้ได้ทันทีว่าทำงานอะไร และ เรียกใช้โดยการผ่านค่าอย่างไร

```
/* Dos Error Codes */
#define EZERO      0 /* Error 0 */
#define EINVFNC    1 /* Invalid function number */
#define ENOFILE    2 /* File not found */
#define ENOPATH    3 /* Path not found */
#define ECONTR     7 /* Memory blocks destroyed */
#define EINVMEM    9 /* Invalid memory block address */
#define EINVENV   10 /* Invalid environment */
#define EINVFMT   11 /* Invalid format */
#define EINVACC   12 /* Invalid access code */
#define EINVDAT   13 /* Invalid data */
#define EINVDRV   15 /* Invalid drive specified */
#define ECURDIR   16 /* Attempt to remove CurDir */
#define ENOTSAM   17 /* Not same device */
#define ENMFILE   18 /* No more files */
```

รูปที่ 3.1 บางส่วนของ errno.h ของ TC2.0 ซึ่งแสดงถึงการใช้ตัวใหญ่ล้วนในการกำหนด (define) ค่าคงที่

- ไม่ควรใช้อักษรตัวเล็กในการเขียนมาโครและบางทีก็รวมถึงการกำหนดชนิดข้อมูลใหม่ด้วย typedef สำหรับ struct ลองอ่านดูใน include file มาตรฐานจะเห็นว่าใช้ใหญ่ล้วนทั้งนั้น เช่น ส่วนหนึ่งของ errno.h และ stdio.h ของ TC 2.0 ที่แสดงไว้ในรูปที่ 3.1 และ 3.2 ตามลำดับ

- ชื่อที่ประกอบด้วยคำหลายคำ ควรแยกคำเหล่านั้นออกจากกันด้วยเครื่องหมายอันเดอร์สกอร์ (underscore (_)) เช่น max_boy หรือ โดยการขึ้นต้นด้วยตัวใหญ่ทุกคำเช่น MaxBoy หรือทั้งสองวิธีรวมกัน คือ Max_Boy

- ชื่อของฟังก์ชันหรือตัวแปรโกลบอล (global) อาจทำให้เด่น ด้วยการขึ้นต้นด้วยอัพเปอร์เคส (Uppercase) เช่น ฟังก์ชัน Get_Token หรือตัวแปร global ชื่อ Total_Boy

- การตั้งชื่อทุกคำ ควรจะมีความแตกต่างกันใน 8 ตัวอักษรแรก เพราะคอมพายเลอร์ (compiler) เก่า ๆ บางตัวจะพิจารณาเฉพาะ 8 ตัวแรกเท่านั้น และถ้าเป็นชื่อที่เป็นเอ็กซ์เทอร์นัลออปเจ็กต์ (external object) ควรจะต่างกันใน 6 ตัวแรกเลย เพราะลิงเคอร์ (linker) บางตัวสนใจเพียงแค่ 6 ตัวแรก ยิ่งไปกว่านั้นไม่ควรตั้งชื่อ 2 ชื่อเหมือนกัน แต่ต่างกันเพียงแค่ใช้อักษรตัวใหญ่หรือตัวเล็ก เพราะนอกจากลิงเคอร์บางตัวจะบอกข้อผิดพลาด คนเรายังแยกแยะความแตกต่างได้ไม่ชัดเจนด้วย

- ในการเขียนโปรแกรมในงานประเภทแอพพลิเคชันหรืองานที่ไม่ใช่โปรแกรมระบบ (system program) ไม่ควรตั้งชื่อขึ้นต้นด้วยอันเดอร์สกอร์ (_) เพราะปกติแล้ว การขึ้นต้นชื่อด้วยอันเดอร์สกอร์จะใช้กับงานโปรแกรมระบบเท่านั้น

– อาจเติมคำขึ้นต้น (prefix) หรือคำลงท้าย (suffix) เข้ากับชื่อตัวแปรเพื่อบอกชนิดข้อมูลบางอย่างเช่น เติมคำลงท้ายว่า _ptr, _int หรือ _str เพื่อบอกว่าชื่อนั้นเป็น pointer, integer หรือ string ตามลำดับ ส่วนการเติมคำขึ้นต้นให้สังเกตได้จากฟังก์ชันมาตรฐานใน string.h เช่น strcmp, strcpy เป็นต้น

– ชื่อบางชื่อควรใช้กับสิ่งของเฉพาะอย่าง เช่น ใช้ i, j และ k กับตัวแปรที่เป็นตัวควบคุมลูป หรือ s มักใช้เป็นตัวแปรชั่วคราวเกี่ยวกับสตริงเป็นต้น และบางครั้งก็พบว่ามีการเติมอักษร x เข้าข้างหลังตัวแปรเหล่านี้เป็น ix, หรือ sx เป็นต้น เพื่อประโยชน์ในการค้นหาคำ มิฉะนั้นเวลาหาคำด้วย editor หรือ grep เพื่อหาตัวแปร i จะไปพบ i ในคำว่า if บ้าง while บ้าง int บ้าง หรือ continue บ้างเป็นต้น

ใช้ typedef ช่วย

ในบางโอกาสที่เราต้องใช้ชนิดตัวแปรที่ซับซ้อน เช่น 'อะเรย์ของตัวชี้ไปยังฟังก์ชันซึ่งคืนค่าเป็น int' (array of function returning int) ก็ควรจะใช้ typedef ช่วย ความจริง เราสามารถใช้ typedef กับข้อมูลทุกชนิดรวมทั้ง int หรือ long แต่ในทางปฏิบัติแล้วก็ไม่ควร เพราะจะเป็นการเพิ่มภาระในการจำ แต่ก็พอมีข้อยกเว้นบ้าง จะกล่าวในตอนหลัง

สำหรับชนิดข้อมูลที่ซับซ้อนแล้ว การใช้ typedef จะช่วยให้อ่านง่ายขึ้น ดังรูปที่ 4 บรรทัดแรกคือ "f เป็นอะเรย์ของตัวชี้ไปยังฟังก์ชันที่คืนค่าเป็น integer" ซึ่งการเขียนเช่นนี้ทำให้อ่านยาก แต่ในการประกาศตัวแปร g ในบรรทัดต่อ ๆ ไป เราจะรู้ว่า int_func_ptr เป็นชนิดข้อมูลของตัวชี้ไปยังฟังก์ชันที่คืนค่าเป็น integer ในบรรทัดที่ 2 บอกว่า func_array เป็นอะเรย์ของ int_func_ptr และในบรรทัดสุดท้ายก็บอกให้เรารู้ว่า g เป็นข้อมูลชนิด func_array หรือ "อะเรย์ของ (ตัวชี้ไปยังฟังก์ชันที่คืนค่าเป็น integer)"

```
/* Definition of the control structure for streams */
typedef struct {
    short          level;   /* fill/empty level of buffer */
    unsigned       flags;   /* File status flags */
    char           fd;      /* File descriptor */
    unsigned char  hold;    /* Ungetc char if no buffer */
    short          bsize;   /* Buffer size */
    unsigned char  *buffer; /* Data transfer buffer */
    unsigned char  *curp;   /* Current active pointer */
    unsigned       istemp;  /* Temporary file indicator */
    short          token;   /* Used for validity checking */
} FILE;                     /* This is the FILE object */
```

รูปที่ 3.2 บางส่วนของ stdio.h ของ TC2.0 ซึ่งแสดงถึงการใช้ตัวใหญ่ล้วนในการ typedef structure

```
int          (*f[ ]) ( );
typedef      int   (*int_func_ptr) ( );
typedef      int_func_ptr    func_array[ ];
func_array g;
```

รูปที่ 4 ตัวอย่างแสดงการใช้ typedef เพื่อประกาศตัวแปร g เป็นชนิดเดียวกันกับ f

จะเห็นได้ว่าการใช้ typedef จะทำให้ต้องเขียนโค้ดมากขึ้นแต่ข้อดีก็คือช่วยให้การทำความเข้าใจชนิดของตัวแปรแบ่งออกเป็นขั้นตอน ซึ่งทำให้เข้าใจง่ายขึ้น แถมยังลดความเสี่ยงในการเกิดบักในตอนเขียนด้วย

เลือกใช้ชนิดข้อมูล

ชนิดของข้อมูลจะบอกรูปแบบของการเก็บ, ขนาดและวิธีการของการใช้ข้อมูลนั้น ช่วยให้เรารู้ขอบเขตของข้อมูลที่เก็บได้และการเปลี่ยนชนิดข้อมูล (type conversion) ซึ่งสำคัญต่อการเข้าใจวิธีการทำงานของโปรแกรม

ในการโปรแกรมมิงนั้น เราต้องระวังการใช้ชนิดของข้อมูลให้ดี โดยเฉพาะในภาษา C ซึ่งมีชนิดของข้อมูลพื้นฐาน และกลไกมากมายที่ช่วยสร้างข้อมูลชนิดใหม่ขึ้นมาแต่ C ยังไม่มีกลไกที่เพียงพอในการป้องกันการใช้ข้อมูลโปรแกรมเมอร์อาจใช้ข้อมูลชนิด char ปนกับชนิด int หรือ long ได้ โดยคอมพายเลอร์ภาษาซีจะไม่เตือนและแถมยังทำการเปลี่ยนชนิดข้อมูลให้อีกด้วย ซึ่งเป็นความสะดวกสบายอย่างหนึ่ง แต่นั่นแหละที่ทำให้เกิดปัญหาได้ในการคอนเวิร์สจากใหญ่ไปเล็ก เช่นถ้าเอาข้อมูลที่อยู่ long ไปใส่ใน char โดยเราไม่ได้ระวังเรื่องขอบเขตที่ char รับได้ก็จะทำให้ข้อมูลผิดพลาด

การใช้ typedef ในหัวข้อที่แล้วที่บอกว่ามีข้อยกเว้นบ้างเหมือนกัน ก็เพราะ C ไม่มีข้อมูลมาตรฐานสำหรับตรรก (boolean) การใช้งานเราจึงนิยมใช้ int แทน และการใช้ typedef ชนิด int ให้เป็น boolean หรือ bool หรือ BOOL ก็เป็นที่นิยมกันทั่วไป เพื่อความชัดเจนว่า

integer ตัวนี้ เราจะมองเพียงแค่ว่าจริงหรือเท็จเพียง 2 ค่าเท่านั้น

C เวอร์ชันที่ไม่เก่าจนเกินไปจะมีข้อมูลชนิด void ไว้ให้ใช้กับฟังก์ชัน เพื่อบอกให้รู้ว่าฟังก์ชันนั้นไม่มีการคืนค่าใด ๆ แก่ฟังก์ชันที่เรียกใช้ หรือเทียบเท่าโพรซีเดอร์ ในภาษาปาสคาล ถ้าโปรแกรมเมอร์ไม่ใส่ชนิดหน้าฟังก์ชันคอมพายเลอร์จะมองฟังก์ชันนั้นว่าคืนค่าเป็น integer ให้โดยอัตโนมัติ และความอัตโนมัตินี้แหละที่ทำให้การไม่ใส่ชนิดของฟังก์ชัน อาจเกิดขึ้นได้ทั้งจากความตั้งใจและไม่ได้ตั้งใจ เพราะฉะนั้นถ้าจะให้ดี ก็ควรใส่ void ด้วย กรณีที่เป็นโพรซีเดอร์เพื่อเป็นการเตือนสติเราทางหนึ่ง แต่ถ้าใช้คอมพายเลอร์ภาษาซีที่ไม่มี void ให้ ก็อาจจะใช้ typedef int void เพื่อบอกว่า void เป็นชนิด int ซึ่งอาจจะขัดกับหลักการใช้ typedef

และ C สมัยนี้ก็ยังมีข้อมูลชนิด enum ให้อีกด้วย enum นี้ย่อมาจาก enumeration ซึ่งแปลว่าการแจงนับหรือคล้าย ๆ กับ set ในปาสคาลดังตัวอย่างในรูปที่ 5.1 5 บรรทัดแรกเราใช้ #define เพื่อเป็นการเปลี่ยนการอ้างถึง 0, 1, ..., 4 ให้เป็น ATLAS, BOOK, ..., MAGAZINE และในบรรทัดสุดท้ายจะประกาศตัวแปร current_document ว่าเป็น integer ซึ่งเราก็กำหนดเอาในใจว่าจะเป็น 0 ถึง 4 เท่านั้น การกำหนดแบบนี้ก็ดีอย่างหนึ่งคือ เป็นการเปลี่ยน

```
#define   ATLAS      0
#define   BOOK       1
#define   INDEX      2
#define   JOURNAL    3
#define   MAGAZINE   4
int       current_document;
```

รูปที่ 5.1 ตัวอย่างการใช้ #define เพื่อแจงชนิดข้อมูล

ชื่อตัวเลข 0, 1, ... ให้เป็นสิ่งที่เรากำลังเกี่ยวข้อง แต่ว่าในตอนประกาศตัวแปร current_document จะเห็นว่ายังไม่เกิดภาพพจน์ที่เพียงพอนักดูในรูปที่ 5.2 เราใช้ typedef เพื่อประกาศชนิดของ document_type ว่ามี ATLAS, BOOK, ... และบรรทัดหลังก็บอกว่า current_doc เป็นตัวแปรชนิด document_type การประกาศแบบนี้จะทำให้ผู้อ่าน และรวมทั้งโปรแกรมเมอร์ตัวเองเกิดภาพพจน์ในการใช้ค่าของตัวแปรได้ดีกว่าในรูปที่ 5.1 ซึ่งให้ผลที่เหมือนกัน แต่ใช้ #define

ข้อดีอีกอย่างหนึ่งของการใช้ enum ก็คือในกรณีที่เราต้องเพิ่มแบบแทรก (insert) ชนิดซึ่งเป็นองค์ประกอบใน enum หรือรายชื่อซึ่งเรากำหนดไว้ ถ้าเราใช้ enum จะเพิ่มได้ง่ายกว่ามาก เพราะถ้าเป็นแบบที่กำหนดไว้แล้ว เราต้องมาเรียงลำดับค่าก็กำหนดใหม่ ทำให้วุ่นวาย

```
typedef enum {ATLAS, BOOK, INDEX, JOURNAL, MAGAZINE} document_type;
document_type current_document;
```

รูปที่ 5.2 ตัวอย่างการใช้ enum เพื่อแจ้งชนิดข้อมูลเหมือนกับใน 5.1 แต่สื่อความหมายได้ดีกว่า

เลือกใช้ Control Structure

C จัดว่าเป็น "ภาษาโครงสร้าง" (Structural Programming) ที่สมบูรณ์อยู่แล้ว คือ มีคำสั่งที่ช่วยในการเขียนให้เป็นโครงสร้างอย่างมากมาย ทั้ง for, while, do-while, switch หรือ if-else ซึ่งลักษณะเช่นนี้จะช่วยให้คนอ่านอ่านได้ง่ายเข้าใจเร็ว ถ้ามีการเลือกใช้ที่ถูกต้องเหมาะสม แต่ขณะเดียวกัน หากใช้คอนโทรลสตรัคเจอร์ (control structure) เหล่านี้ไม่ดีแล้ว ก็จะกลายเป็นความยุ่งเหยิง ปวดหัวได้เหมือนกัน

ในรูปที่ 6.1 นั้นเป็นตัวอย่างการเลือกใช้คอนโทรลสตรัคเจอร์ที่ไม่เหมาะสม เมื่อไล่ดูการทำงาน จะพบว่ามีการตั้งต้นค่าแก่ตัวแปร i ตรวจดูถ้าค่า i ไม่เท่ากับ MAX ก็ไปเรียก Process และเพิ่มค่า i ใหม่ แล้วกลับไปตรวจดูใหม่จนกว่า i จะเท่ากับ MAX ถึงจะหลุดจากลูปได้ แต่เมื่อดูตามโฟลว์นี้แล้วเราก็อาจจะ for ลูปแทนได้ง่าย ๆ ดังในรูปที่ 6.2 ก็จะให้ผลเช่นเดียวกัน ซึ่งนักเขียน C จะเข้าใจโครงสร้างในรูปที่ 6.2 ได้ง่ายกว่าในรูป 6.1 มากทีเดียว

สำหรับแนวทางง่าย ๆ ที่น่าคำนึงถึงเวลาเลือกใช้คอนโทรลสตรัคเจอร์สำหรับงานต่าง ๆ เพื่อความอ่านง่ายและถูกต้องของโค้ดได้แก่

- ถ้าจำเป็นต้องทำงานในลูปอย่างน้อย 1 ครั้ง ก็ให้เลือกใช้ do-while มิฉะนั้นก็ให้ใช้ for หรือ while แทน
- ใช้สวิตช์ก็ต่อเมื่อมีตัวเลือกหรือเคสมากกว่า 2 ตัวเลือก ถ้ามีแค่ 1 หรือ 2

```
i = 0;
while (TRUE)
{
    if (MAX = = i) break;
    Process(i);
    i + +;
}
```

รูปที่ 6.1 ตัวอย่างการใช้ control structure ที่ไม่เหมาะสมจะทำให้อ่านยากกว่าในรูปที่ 6.2 ซึ่งทำงานอย่างเดียวกัน

```
for (i = 0; i < MAX: i + +)
    process (i);
```

รูปที่ 6.2 ตัวอย่างใช้ control structure ที่เหมาะสมกว่าในรูปที่ 6.1

ควรใช้ if-else แทนจะดีกว่า

– ไม่ควรใช้ goto เพราะจะทำให้โฟลว์ของโปรแกรมกระโดดมั่ว C มีคอนโทรลสตรัคเจอร์อื่นให้เลือกใช้อย่างเหลือเฟือแล้ว ถ้าไม่จำเป็นจริง ๆ ก็ไม่ควรอย่างยิ่งที่จะใช้ goto

– เลี่ยงการใช้ continue ถ้าไม่จำเป็น เหตุผลก็คล้าย ๆ กับ goto แต่ว่า continue ไม่ทำให้โค้ดเละเทอะเท่า goto เราสามารถเลี่ยง continue ในลูป for, while หรือ do-while ได้โดยการใช้ if ที่ดีพอ

– เลี่ยงการใช้ break หรือ return ก่อนจบ block คำสั่งถ้าไม่จำเป็น (อันนี้ไม่นับการใช้ break กับ switch-case นะ) ซึ่งการเลี่ยงนี้มักจะทำให้เกิด flag ทดสอบเพิ่มขึ้นมาบ้าง แต่ถ้าเป็นอัลกอริทึมที่ซับซ้อนอยู่แล้ว การใช้ break หรือ return อาจช่วยลดความซับซ้อนลงได้บ้าง อันนี้ต้องชั่งน้ำหนักดูให้ดีเองก็แล้วกัน

เขียนให้เป็นโครงสร้าง

ในรูปที่ 7.1 จะเห็นว่าการเขียน if-else อย่างต่อเนื่องเป็นแถวเดียวกันหมด ทำให้อ่านยากกว่าในรูปที่ 7.2 ซึ่งให้ผลอย่างเดียวกัน แต่เขียนโดยมีการใช้ tab จัด indent และใช้ช่องว่างแบ่งคำด้วย (นอกเหนือจากการตั้งชื่อ maxstudent ใหม่เป็น Max_Student ตามหัวข้อแรก ๆ ที่ผ่านมา) ซึ่งถ้าเขียนเป็นโค้ดยาว ๆ แบบอัดกันแน่นแบบในรูปที่ 7.1 รับรองว่าจะได้โค้ดที่ไม่น่าอ่านที่สุด

```
if (id>maxstudent) {err=ERR_OVF; return -1;}
else <err=0; return 0;}
```

รูปที่ 7.1 ตัวอย่างการเขียนที่ไม่มีการแบ่งช่องว่างและบรรทัดที่เหมาะสมทำให้อ่านยาก

เรื่องการจัดรูปแบบการเขียนที่นิยมกันสามารถแบ่งได้เป็น 4 แบบด้วยกัน ซึ่งแต่ละคนที่ใช้แบบไหนก็จะบอกว่าแบบนั้นดี อ่านง่าย อันนี้ก็แล้วแต่ว่าคนไหนจะเลือกใช้แบบไหนก็ตามใจ 4 แบบที่ว่าก็อยู่ในรูปที่ 8

หลักการเกี่ยวกับการเติม tab, blank หรือ newline (ซึ่งรวมเรียกว่า white space character) นี้มีคร่าว ๆ ได้แก่

– จัด indent ของชุดคำสั่งในระดับเดียวกันให้ตรงคอลัมน์

– จัดกลุ่มคำสั่งซึ่งรองลงไป ให้เยื้องไปทางขวา 1 tab อย่างเช่น รูปแบบที่ดีของ if-else คือในรูปที่ 9.1

– ใช้ blank หรือช่องว่างในการจัดแบ่งกลุ่มคำในประโยค เพื่อบอกหน้าที่การทำงานย่อย ๆ เช่น ไม่เขียนนิพจน์อย่างในรูปที่ 9.2 แต่ควรจะเขียนแทนแบบในรูป 9.3 เป็นต้น ซึ่งช่วยให้ผู้อ่านเข้าใจขั้นตอนการคำนวณได้รวดเร็ว

– ใช้ newline แบ่งกลุ่มคำสั่งออกเป็นกลุ่มเล็ก ๆ ซึ่งทำงานย่อยอย่างใดอย่างหนึ่ง เช่นไม่ควรเขียนอย่างในรูปที่ 10.1 แต่ควรใช้ newline ช่วยแบ่ง อย่างในรูปที่ 10.2 ประโยชน์จะเห็นได้ชัดยิ่งขึ้น เมื่อโค้ดยาวขึ้น

– ในคำสั่ง for ซึ่งประกอบไปด้วย 3 ส่วนคือ ส่วนตั้งค่าเริ่มต้น, ส่วนเปรียบเทียบ, และส่วนการเปลี่ยนแปลง ถ้าส่วนไหนไม่มี เราก็ควรจะเติมคอมเมนต์บอกไว้ว่าส่วนนี้ไม่มีจริง ๆ ไม่ใช่ว่าเราลืมเขียนหรือเผลอลบทิ้ง เช่น

```
for (/* old value of i */i i < 10 ; i + +)
```

– ในการทำลูปอนันต์หรือลูปที่วนไม่รู้จบ (forever loop) ที่เรามักใช้กันว่า "for (;;) ..." ถ้าเขียนในรูป "while (TRUE)..." ก็จะชัดเจนขึ้น แต่ที่จริงแล้วจะพบงานประเภททำไม่รู้จบอย่างนี้น้อย และการกระโดดออกจากลูปประเภทนี้โดยการใช้รีเทอร์นหรือเบรคก็จะขัดกับหลักการเลือกใช้คอนโทรลสตรัคเจอร์อยู่แล้ว ดังนั้นการใช้งานจริงก็จะไม่พบรูปแบบของลูปอนันต์บ่อยนัก

– C มีการใช้คำสั่งที่ไม่มีการทำงานใด ๆ คือ ; ลอย ๆ ซึ่งพบในตัวลูป บางครั้งคนอ่านอาจนึกว่า semicolon (;)

```
if (id > Max_Student)
{
        err = ERR_OVF;
        return -1;
} else {
        err = 0;
        return 0;
}
```

รูปที่ 7.2 ตัวอย่างการเขียนที่มีการใช้ white space character ที่ดี ช่วยให้อ่านโค้ดง่ายกว่าในรูปที่ 7.1 ซึ่งทำงานเหมือนกัน

```
Pascal Style
        while (test)
              {
              statements;
              }
Modified Pascal Style
        while (test)
        {
              statements;
        }
Kernighan and Rithie Style
        while (test) {
              statements;
        }
Modified Kernighan and Rithie Style
        while (test) {
              statements;
              }
```

รูปที่ 8 รูปแบบการจัด indent ที่เป็นมาตรฐาน 4 แบบ

```
if (condition)
  statement_1
else
  statement_2;
```

รูปที่ 9.1 รูปแบบหนึ่งของการเขียน if-else statement ที่ดี

```
a=b*c-d/e+3.14159*f/g;
```

รูปที่ 9.2 ตัวอย่างการเขียนนิพจน์ที่ไม่ใช้ blank ช่วยแบ่งกลุ่ม ทำให้อ่านยาก

```
a = b*c - d/e + 3.14159 * f/g ;
```

รูปที่ 9.3 ตัวอย่างการเขียนนิพจน์ที่ใช้ blank ช่วยแบ่งกลุ่มซึ่งช่วยให้อ่านง่ายขึ้น

นี้ เขียนเกินมา อะไรทำนองนั้น แต่ถ้าเกิดตัดออกไป รับรองการทำงานผิดจากเดิมแน่ นี่แหละที่บอกว่า C นั้นมีสัญลักษณ์ที่สร้างความผิดพลาดได้ง่าย วิธีป้องกันคือ ใส่คอมเมนต์บอกว่านี้เป็นลูปที่ไม่ทำอะไรในตัวลูป เช่น

```
while( *dest++ = *src++ )
    /* null loop body */;
```

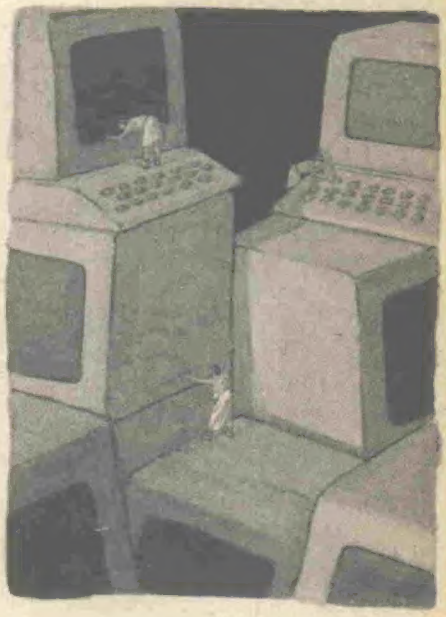
– ระวังการละเบรคในสวิตช์ให้ดี ซึ่งหลายคนมักจะลืมใส่ ทำให้เกิดบักได้ แต่ในบางกรณีที่ตั้งใจไม่ใส่ ก็อาจทำให้ผู้อ่านคิดเป็นว่าลืมก็ได้ ดังนั้นควรจะบอกไว้ด้วยคอมเมนต์เช่น

```
...
case 'x' : case_x_statement;
           /* no break ! -- common
           x and y processing */
case 'y' : case_y_statement ;
           break;
```

ซึ่งการเขียนลักษณะนี้ ถ้าพอเลี่ยงได้ก็เลี่ยงดีกว่า เพราะถ้าเกิดสลับตำแหน่งเคสเหล่านี้ ทำให้ y ไม่ได้อยู่ถัดจาก x อาจจะทำให้เกิดบักได้

Comment

ก็เพราะ C เป็นภาษาที่เต็มไปด้วยสัญลักษณ์ ในบางครั้งถึงจะเขียนตามหลักที่กล่าวมาข้างต้นแค่ไหนซอร์สโค้ดที่ได้ก็ยังอ่านยากอยู่ เนื่องจากความซับซ้อนของขั้นตอนวิธี (algorithm) การเติมคอมเมนต์หรือคำอธิบายประกอบโปรแกรม จะช่วยในส่วนนี้ได้ แต่ว่าโปรแกรมเมอร์หลายคนเขียนคอมเมนต์ที่ดี



ไม่เป็น ก็เพราะยังไม่มีมาตรฐานอะไรชัดเจนบอกว่าเขียนอย่างไรถึงจะเรียกว่าดี

สมมติเขียนโปรแกรมขึ้นมาโปรแกรมหนึ่ง อย่างน้อยก็น่าจะบอกว่าโปรแกรมนี้มีเป้าหมาย (goal) อย่างไร, ทำอย่างไรให้ได้ตามนั้น (plan), บอกคร่าว ๆ ว่าโปรแกรมแต่ละส่วนทำอะไร มีบทบาทอย่างไร (object's role), และทำไมถึงได้ทำอย่างนั้น (object's purpose)

Goal

บอกให้รู้ถึงเป้าหมายของโปรแกรมหรือส่วนย่อยของโปรแกรม ให้รู้ว่าทำขึ้นมาเพื่ออะไร เช่น เขียนโปรแกรมหนึ่งเพื่อแสดงรายชื่อพนักงานในบริษัท

```
scanf ("%s", name);
idx_name = Search_Name (Student_List, name);
if (Process_Name (Student_List, idx_name) == TRUE)
      return Menu (NAME, idx_name);
scanf ("%d", &age);
idx_age = Search_Age (Student_List, age);
if (Process_Age (Student_List, idx_age) == TRUE)
      return Menu (AGE, idx_age);
```

รูปที่ 10.1 แสดงการเขียนที่ต่อเนื่องโดยไม่ใช้ newline

```
scanf ("%s", name);

idx_name = Search_Name (Student_List, name);
if (Process_Name (Student_List, idx_name) == TRUE
      return Menu (NAME, idx_name);

scanf ("%d", &age);
idx_age = Search_Age (Student_List, age);
if (Process_Age (Student_List, idx_age) == TRUE)
      return Menu (AGE, idx_age);
```

รูปที่ 10.2 ตัวอย่างการใช้ newline ช่วยแบ่งกลุ่มคำสั่ง

```
/*****************************************************************
                    --- Function Header Comment ---
                  Plan:  Part 1: First process step
                         Part 2: Second process step
                                ...
                         Part k:   K-th process step
*****************************************************************/
void func_name (argument)
{
      /* Part 1: First process step */
      < code for the first process step >

      /* Part 2: Second process step */
      < code for the second process step >

      /* Part k: K-th process step */
      < code for the k-th process step >
} /* end function definition */
```

รูปที่ 11 รูปแบบการเขียนแผนงาน (plan) สำหรับฟังก์ชัน

```
#define MAX_BRANCH 500 /* maximum number of bank's branch */
```

รูปที่ 12.1 ตัวอย่างการอธิบายค่าคงตัวที่ define ไว้

```
/* ABS_DBL -- Absolute values of doubles        */
/* must be computed in the inner loop of the    */
/* ... functions. These loops must be optimized */
/* to achieve adequate response time            */
#define ABS_DBL(x) ( ( (x) >=0.0 ) ? (x) : -(x) )
```

รูปที่ 12.2 ตัวอย่างการเขียนอธิบาย macro

เรียงตามอายุการทำงานจากมากไปน้อย เป็นต้น เราควรจะบอกเป้าหมายนี้ไว้ในส่วนหัวของฟังก์ชันหรือต้นไฟล์ เพื่อบอกเป้าหมายของฟังก์ชันหรือไฟล์นั้น

Plan

เป็นโครงร่างคร่าว ๆ ที่บอกให้เรารู้ถึงขั้นตอนการทำงานเพื่อให้ได้ถึงเป้าหมาย (goal) นั้น ๆ แปลน (plan) หรือแผนงานนี้จะแบ่งและสรุปโค้ด ให้เป็นส่วนย่อย ๆ ที่ง่ายต่อการทำความเข้าใจ เราควรเขียนแผนงานไว้ 2 ที่ คือ ส่วนหัวของฟังก์ชันและภายในตัวฟังก์ชัน ในส่วนหัวของฟังก์ชันจะเขียนแผนทุกขั้นตอนต่อเนื่องกัน ส่วนในตัวฟังก์ชันก็จะแบ่งแผนย่อย ๆ แทรกไว้ก่อนถึงโค้ดส่วนที่ทำงานตามแผนนั้น ๆ ดังแสดงในรูปที่ 11

ข้อดีของการเขียนแผนงานคือ ช่วยอธิบายให้รู้ว่าฟังก์ชันนั้น ๆ ทำงานให้ได้ตามเป้าหมายได้อย่างไร, ช่วยให้รู้ขอบเขตการทำงานของฟังก์ชัน, ช่วยแบ่งย่อยฟังก์ชันออกเป็นส่วน ๆ ง่ายต่อการทำความเข้าใจ, และสุดท้ายคือเป็นคล้าย ๆ กับสารบัญของฟังก์ชันนั่นเอง

Role and Purpose

ส่วนนี้จะบอกให้รู้ว่าแต่ละส่วนของโปรแกรม มีบทบาทและวัตถุประสงค์อย่างไร โดยเฉพาะอย่างยิ่งสำหรับวัตถุที่ตั้งชื่อให้สื่อความหมายชัดเจนได้ยาก เราควรจะอธิบายด้วยคอมเมนต์ไว้ที่คำประกาศวัตถุนั้นเลย ผู้อ่านจะได้มองเป็นภาพก่อนที่จะพบการใช้งานจริง ๆ ยกตัวอย่างเช่น

1. define การใช้ #define กำหนดค่าคงตัว เราควรจะบอกว่าค่าคงที่ตัวนั้นใช้งานอย่างไรในโปรแกรม เช่นในรูปที่ 12.1

2. macro ปกติเราจะใช้มาโคร (macro) แทนการเขียนเป็นฟังก์ชันก็เพื่อประสิทธิภาพของการทำงาน จะได้ไม่ต้องเสียเวลาในการส่งพารามิเตอร์และเข้าออกฟังก์ชัน แต่ว่าการใช้มาโครนี้จะต้องระวังเรื่องของผลกระทบข้างเคียง (side effect) ที่อาจเกิดได้ ในตอนที่เราประกาศมาโครก็ควรจะอธิบายการใช้งานมันด้วย ดังรูปที่ 12.2

3. variable ควรจะอธิบายว่าตัวแปรเก็บอย่างไร, เก็บทำไมและมีความสำคัญอย่างไร ดีกว่าไปบอกว่าเก็บค่าอะไรอยู่ ดังรูปที่ 12.3

4. parameter บอกบทบาทและวัตถุประสงค์ของ parameter แต่ละตัว และควรจะบอกด้วยว่าแต่ละตัวนั้นเป็นชนิดส่งค่าเข้าฟังก์ชันอย่างเดียว, ผ่านค่าออกฟังก์ชันอย่างเดียว หรือทั้ง 2 อย่าง ดังรูปที่ 12.4

นอกจากจะเขียนคอมเมนต์บอกเป้าหมาย, แผนงาน, บทบาทและจุดมุ่งหมายของวัตถุต่าง ๆ แล้ว ยังเราควรบอกผลกระทบข้างเคียงด้วยถ้ามีและรวมทั้งคอมเมนต์ของฟังก์ชันด้วย

```
BRANCH    Branch [MAX_BRANCH];   /* Array of structure containing      */
                                 /* fundamental information of         */
                                 /* every branches of ... bank         */
int       Branch_Index;          /* Index to point to the current      */
                                 /* considered branch -- ranged from   */
                                 /* 0 to MAX_BRANCH-1                  */
```

รูปที่ 12.3 ตัวอย่างการอธิบายตัวแปร (variables)

```
int Summarize_Branch (branch, mode, ret_struct)
BRANCH          *branch;        /* in/out: branch to be summarized */
int             mode;           /*  in:  mode  of  summary
RET             *ret_struct     /* out: return code structure      */
{
    ...
}
```

รูปที่ 12.4 ตัวอย่างการเขียนอธิบาย parameter

Items	ตำแหน่งที่ว่าง	อธิบาย
Program Goal	ที่ส่วนต้นของฟังก์ชันหรือโมดูล	เป้าหมายการทำงานของฟังก์ชันหรือโมดูลนั้น
Program Plan	ส่วนต้นและภายในตัวฟังก์ชัน	บอกโครงร่างกายทำงานเป็นขั้น ๆ
Object role	ตำแหน่งที่ประกาศ object	หน้าที่บทบาทต่อการทำงานของโปรแกรม
Object purpose	ตำแหน่งที่ประกาศ object	สาเหตุของหน้าที่และบทบาทนั้น
Side-effect Information	ตำแหน่งที่ประกาศวัตถุที่เกิด side effect และจุดที่เกิด	อธิบายผลกระทบที่เกิด

รูปที่ 13 สรุปการเขียนคอมเมนต์

ผลกระทบข้างเคียงหมายถึงการทำงานของส่วนหนึ่งส่วนใดของโปรแกรมที่มีผลกระทบต่อข้อมูลหรือสถานะใด ๆ นอกขอบเขต คือส่วนที่ไม่เป็นโลคอล เช่นฟังก์ชันหนึ่งไปแก้ไขข้อมูลโกลบอลตัวหนึ่ง ถ้ามีผลกระทบข้างเคียงอย่างนี้มาก ๆ จะทำให้ดูแลปรับปรุงโค้ดยาก เพราะมีได้หลายฟังก์ชันนั้นที่สามารถแก้ค่าโกลบอลตัวเดียวกัน แต่ก็มีข้อดีที่ทำให้โปรแกรมที่ได้มีประสิทธิภาพและสะดวก ไม่ต้องส่งพารามิเตอร์มากมาย จึงเป็น tradeoff กัน

สำหรับการเขียนอธิบายผลกระทบข้างเคียงนี้จะต้องบอก 1) วัตถุที่ถูกกระทบ 2) ส่วนของโปรแกรมที่ทำให้เกิดผลกระทบขึ้น 3) ผลที่เกิดขึ้น และ 4) ทำไมถึงเกิดผลกระทบนี้ขึ้น ซึ่งควรเขียนคอมเมนต์สำหรับผลกระทบข้างเคียงนี้ไว้ 3 ที่ด้วยกันคือ ที่หัวของฟังก์ชันที่ทำให้เกิด, ที่คำประกาศของ

วัตถุที่จะเกิดผลกระทบ, และจุดที่เรียกฟังก์ชันที่ทำให้เกิดผลกระทบนั้น

และสุดท้ายคือคอมเมนต์ของฟังก์ชัน ควรจะบอกถึงวัตถุประสงค์ของฟังก์ชันว่าเขียนขึ้นเพื่อใช้ทำอะไร, รีเทอร์นโค้ดและวัตถุประสงค์ที่คืนค่านั้น, แผนงานดังที่ได้กล่าวก่อนหน้านี้แล้ว, และหมายเหตุที่จะอธิบายเพิ่มเติมอะไร เช่น เทคนิคพิเศษที่ฟังก์ชันที่ใช้เป็นต้นก็มาเขียนไว้ที่นี้

สรุปการเขียนคอมเมนต์ง่าย ๆ ไว้ในตารางในรูปที่ 13

บทส่งท้าย

ที่จริงเรื่องของการเขียนคอมเมนต์ยังมีอีกมาก และยังมีเรื่องของการเขียนเอกสารประกอบโปรแกรมอีกด้วย ซึ่งเป็นเรื่องที่ยาวพอดู แต่ตอนนี้อยากจะขอหยุดไว้แค่นี้พอ เพราะถ้าทำได้เท่านี้ก็จะได้โค้ดที่อ่านง่ายขึ้นพอสมควรไว้มีโอกาสจะเขียนเรื่องที่เหลือมาลงต่อ สวัสดีครับ